Active Record Query Interface

This guide covers different ways to retrieve data from the database using Active Record. By referring to this guide, you will be able to:

Find records using a variety of methods and conditions

Specify the order, retrieved attributes, grouping, and other properties of the found records

Use eager loading to reduce the number of database queries needed for data retrieval

Use dynamic finders methods

Check for the existence of particular records

Perform various calculations on Active Record models

Run EXPLAIN on relations

Chapters

This Guide is based on Rails 3.0. Some of the code shown here will not work in other versions of Rails.

If you're used to using raw SQL to find database records, then you will generally find that there are better ways to carry out the same operations in Rails. Active Record insulates you from the need to use SQL in most cases.

Code examples throughout this guide will refer to one or more of the following models:

All of the following models use `id` as the primary key, unless specified otherwise.

```
class

Client < ActiveRecord::Base

has_one
:address

has_many
:orders

has_and_belongs_to_many
:roles
end
```

```
class

Address < ActiveRecord::Base

belongs_to
:client
end
```

```
class

Order < ActiveRecord::Base

belongs_to
:client
```

```
,
:counter_cache

=>
true
end



class

Role < ActiveRecord::Base

has_and_belongs_to_many
:clients
end
```

Active Record will perform queries on the database for you and is compatible with most database systems (MySQL, PostgreSQL and SQLite to name a few). Regardless of which database system you're using, the Active Record method format will always be the same.

1 Retrieving Objects from the Database

To retrieve objects from the database, Active Record provides several finder methods. Each finder method allows you to pass arguments into it to perform certain queries on your database without writing raw SQL.

The methods are:

- `where`
- `select`
- `group`
- `order`
- `reorder`
- `reverse_order`
- `limit`
- `offset`
- `joins`
- `includes`
- `lock`
- `readonly`
- `from`
- `having`

All of the above methods return an instance of `ActiveRecord::Relation`.

The primary operation of `Model.find(options)` can be summarized as:

- Convert the supplied options to an equivalent SQL query.
- Fire the SQL query and retrieve the corresponding results from the database.
- Instantiate the equivalent Ruby object of the appropriate model for every resulting row.
- Run `after_find` callbacks, if any.

1.1 Retrieving a Single Object

Active Record provides five different ways of retrieving a single object.

1.1.1 Using a Primary Key

Using `Model.find(primary_key)`, you can retrieve the object corresponding to the specified *primary key* that matches any supplied options. For example:

```
# Find the client with primary key (id) 10.
client = Client.find(
10
)
# => #<Client id: 10, first_name: "Ryan">
```

The SQL equivalent of the above is:

```
SELECT
```

```
*
FROM

clients
WHERE

(clients.id = 10) LIMIT 1
```

`Model.find(primary_key)` will raise an `ActiveRecord::RecordNotFound` exception if no matching record is found.

1.1.2 `first`

`Model.first` finds the first record matched by the supplied options, if any. For example:

```
client = Client.first
# => #<Client id: 1, first_name: "Lifo">
```

The SQL equivalent of the above is:

```
SELECT

*
FROM

clients LIMIT 1
```

`Model.first` returns `nil` if no matching record is found. No exception will be raised.

1.1.3 `last`

`Model.last` finds the last record matched by the supplied options. For example:

```
client = Client.last
# => #<Client id: 221, first_name: "Russel">
```

The SQL equivalent of the above is:

```
SELECT

*
FROM

clients
ORDER

BY

clients.id
DESC

LIMIT 1
```

`Model.last` returns `nil` if no matching record is found. No exception will be raised.

1.1.4 `first!`

`Model.first!` finds the first record. For example:

```
client = Client.first!
# => #<Client id: 1, first_name: "Lifo">
```

The SQL equivalent of the above is:

```
SELECT

*
```

```
FROM

clients LIMIT 1
```

`Model.first!` raises `RecordNotFound` if no matching record is found.

1.1.5 `last!`

`Model.last!` finds the last record. For example:

```
client = Client.last!
# => #<Client id: 221, first_name: "Russel">
```

The SQL equivalent of the above is:

```
SELECT

*
FROM

clients
ORDER

BY

clients.id
DESC

LIMIT 1
```

`Model.last!` raises `RecordNotFound` if no matching record is found.

1.2 Retrieving Multiple Objects

1.2.1 Using Multiple Primary Keys

`Model.find(array_of_primary_key)` accepts an array of *primary keys*, returning an array containing all of the matching records for the supplied *primary keys*. For example:

```
# Find the clients with primary keys 1 and 10.
client = Client.find([
1
,
10
])
# Or even Client.find(1, 10)
# => [#<Client id: 1, first_name: "Lifo">, #<Client id: 10, first_name: "Ryan">]
```

The SQL equivalent of the above is:

```
SELECT

*
FROM

clients
WHERE

(clients.id
IN

(1,10))
```

`Model.find(array_of_primary_key)` will raise an `ActiveRecord::RecordNotFound` exception unless a matching record is found for **all** of the supplied primary keys.

1.3 Retrieving Multiple Objects in Batches

We often need to iterate over a large set of records, as when we send a newsletter to a large set of users, or when we export data.

This may appear straightforward:

```
# This is very inefficient when the users table has thousands of rows.
User.all.
each

do

|user|

NewsLetter.weekly_deliver(user)
end
```

But this approach becomes increasingly impractical as the table size increases, since `User.all.each` instructs Active Record to fetch *the entire table* in a single pass, build a model object per row, and then keep the entire array of model objects in memory. Indeed, if we have a large number of records, the entire collection may exceed the amount of memory available.

Rails provides two methods that address this problem by dividing records into memory-friendly batches for processing. The first method, `find_each`, retrieves a batch of records and then yields *each* record to the block individually as a model. The second method, `find_in_batches`, retrieves a batch of records and then yields *the entire batch* to the block as an array of models.

The `find_each` and `find_in_batches` methods are intended for use in the batch processing of a large number of records that wouldn't fit in memory all at once. If you just need to loop over a thousand records the regular find methods are the preferred option.

1.3.1 `find_each`

The `find_each` method retrieves a batch of records and then yields *each* record to the block individually as a model. In the following example, `find_each` will retrieve 1000 records (the current default for both `find_each` and `find_in_batches`) and then yield each record individually to the block as a model. This process is repeated until all of the records have been processed:

```
User.find_each
do

|user|

NewsLetter.weekly_deliver(user)
end
```

1.3.1.1 Options for `find_each`

The `find_each` method accepts most of the options allowed by the regular `find` method, except for `:order` and `:limit`, which are reserved for internal use by `find_each`.

Two additional options, `:batch_size` and `:start`, are available as well.

`:batch_size`

The `:batch_size` option allows you to specify the number of records to be retrieved in each batch, before being passed individually to the block. For example, to retrieve records in batches of 5000:

```
User.find_each(
:batch_size

=>
5000
)
do

|user|

NewsLetter.weekly_deliver(user)
end
```

`:start`

By default, records are fetched in ascending order of the primary key, which must be an integer. The `:start` option allows you to configure the first ID of the sequence whenever the lowest ID is not the one you need. This would be useful, for example, if you wanted to resume an interrupted batch process, provided you saved the last processed ID as a checkpoint.

For example, to send newsletters only to users with the primary key starting from 2000, and to retrieve them in batches of 5000:

```
User.find_each(
:start

=>
2000
,
:batch_size

=>
5000
)
do

|user|

NewsLetter.weekly_deliver(user)
end
```

Another example would be if you wanted multiple workers handling the same processing queue. You could have each worker handle 10000 records by setting the appropriate `:start` option on each worker.

The `:include` option allows you to name associations that should be loaded alongside with the models.

1.3.2 `find_in_batches`

The `find_in_batches` method is similar to `find_each`, since both retrieve batches of records. The difference is that `find_in_batches` yields *batches* to the block as an array of models, instead of individually. The following example will yield to the supplied block an array of up to 1000 invoices at a time, with the final block containing any remaining invoices:

```
# Give add_invoices an array of 1000 invoices at a time
Invoice.find_in_batches(
:include

=>
:invoice_lines
)
do

|invoices|

export.add_invoices(invoices)
end
```

The `:include` option allows you to name associations that should be loaded alongside with the models.

1.3.2.1 Options for `find_in_batches`

The `find_in_batches` method accepts the same `:batch_size` and `:start` options as `find_each`, as well as most of the options allowed by the regular `find` method, except for `:order` and `:limit`, which are reserved for internal use by `find_in_batches`.

2 Conditions

The `where` method allows you to specify conditions to limit the records returned, representing the `WHERE`-part of the SQL statement. Conditions can either be specified as a string, array, or hash.

2.1 Pure String Conditions

If you'd like to add conditions to your find, you could just specify them in there, just like `Client.where("orders_count = '2'")`. This will find all clients where the `orders_count` field's value is 2.

Building your own conditions as pure strings can leave you vulnerable to SQL injection exploits. For example, `Client.where("first_name LIKE '%#{params[:first_name]}%'")` is not safe. See the next section for the preferred way to handle conditions using an array.

2.2 Array Conditions

Now what if that number could vary, say as an argument from somewhere? The find would then take the form:

```
Client.where(
"orders_count = ?"
, params[
:orders
])
```

Active Record will go through the first element in the conditions value and any additional elements will replace the question marks `(?)` in the first element.

If you want to specify multiple conditions:

```
Client.where(
"orders_count = ? AND locked = ?"
, params[
:orders
],
false
)
```

In this example, the first question mark will be replaced with the value in `params[:orders]` and the second will be replaced with the SQL representation of `false`, which depends on the adapter.

This code is highly preferable:

```
Client.where(
"orders_count = ?"
, params[
:orders
])
```

to this code:

```
Client.where(
"orders_count = #{params[:orders]}"
)
```

because of argument safety. Putting the variable directly into the conditions string will pass the variable to the database **as-is**. This means that it will be an unescaped variable directly from a user who may have malicious intent. If you do this, you put your entire database at risk because once a user finds out he or she can exploit your database they can do just about anything to it. Never ever put your arguments directly inside the conditions string.

For more information on the dangers of SQL injection, see the **Ruby on Rails Security Guide**.

2.2.1 Placeholder Conditions

Similar to the `(?)` replacement style of params, you can also specify keys/values hash in your array conditions:

```
Client.where(
"created_at >= :start_date AND created_at <= :end_date"
,

{
:start_date

=> params[
:start_date
],
:end_date

=> params[
:end_date
]})
```

This makes for clearer readability if you have a large number of variable conditions.

2.2.2 Range Conditions

If you’re looking for a range inside of a table (for example, users created in a certain timeframe) you can use the conditions option coupled with the `IN` SQL statement for this. If you had two dates coming in from a controller you could do something like this to look for a range:

```
Client.where(
:created_at

=> (params[
:start_date
].to_date)..(params[
:end_date
].to_date))
```

This query will generate something similar to the following SQL:

```
SELECT

"clients"
.*
FROM

"clients"

WHERE

(
"clients"
.
"created_at"

BETWEEN

'2010-09-29'

AND

'2010-11-30'
)
```

2.3 Hash Conditions

Active Record also allows you to pass in hash conditions which can increase the readability of your conditions syntax. With hash conditions, you pass in a hash with keys of the fields you want conditionalised and the values of how you want to conditionalise them:

Only equality, range and subset checking are possible with Hash conditions.

2.3.1 Equality Conditions

```
Client.where(
:locked

=>
true
)
```

The field name can also be a string:

```
Client.where(
'locked'

=>
true
)
```

2.3.2 Range Conditions

The good thing about this is that we can pass in a range for our fields without it generating a large query as shown in the preamble of this section.

```
Client.where(
:created_at
=> (
Time
.now.midnight -
1
.day)..
Time
.now.midnight)
```

This will find all clients created yesterday by using a BETWEEN SQL statement:

```
SELECT

*
FROM

clients
WHERE

(clients.created_at
BETWEEN

'2008-12-21 00:00:00'

AND

'2008-12-22 00:00:00'
)
```

This demonstrates a shorter syntax for the examples in **Array Conditions**

2.3.3 Subset Conditions

If you want to find records using the `IN` expression you can pass an array to the conditions hash:

```
Client.where(
:orders_count

=> [
1
,
3
,
5
])
```

This code will generate SQL like this:

```
SELECT

*
FROM

clients
WHERE

(clients.orders_count
IN

(1,3,5))
```

3 Ordering

To retrieve records from the database in a specific order, you can use the `order` method.

For example, if you're getting a set of records and want to order them in ascending order by the `created_at` field in your table:

```
Client.order(
"created_at"
)
```

You could specify ASC or DESC as well:

```
Client.order(
"created_at DESC"
)
# OR
Client.order(
"created_at ASC"
)
```

Or ordering by multiple fields:

```
Client.order(
"orders_count ASC, created_at DESC"
)
```

4 Selecting Specific Fields

By default, `Model.find` selects all the fields from the result set using `select *`.

To select only a subset of fields from the result set, you can specify the subset via the `select` method.

If the `select` method is used, all the returning objects will be **read only**.

For example, to select only `viewable_by` and `locked` columns:

```
Client.select(
"viewable_by, locked"
)
```

The SQL query used by this find call will be somewhat like:

```
SELECT

viewable_by, locked
FROM

clients
```

Be careful because this also means you're initializing a model object with only the fields that you've selected. If you attempt to access a field that is not in the initialized record you'll receive:

```
ActiveModel::MissingAttributeError: missing attribute: <attribute>
```

Where `<attribute>` is the attribute you asked for. The `id` method will not raise the `ActiveRecord::MissingAttributeError`, so just be careful when working with associations because they need the `id` method to function properly.

If you would like to only grab a single record per unique value in a certain field, you can use `uniq`:

```
Client.select(
:name
).uniq
```

This would generate SQL like:

```
SELECT

DISTINCT

name
```

```
FROM

clients
```

You can also remove the uniqueness constraint:

```
query = Client.select(
:name
).uniq
# => Returns unique names

query.uniq(
false
)
# => Returns all names, even if there are duplicates
```

5 Limit and Offset

To apply `LIMIT` to the SQL fired by the `Model.find`, you can specify the `LIMIT` using `limit` and `offset` methods on the relation.

You can use `limit` to specify the number of records to be retrieved, and use `offset` to specify the number of records to skip before starting to return the records. For example

```
Client.limit(
5
)
```

will return a maximum of 5 clients and because it specifies no offset it will return the first 5 in the table. The SQL it executes looks like this:

```
SELECT

*
FROM

clients LIMIT 5
```

Adding `offset` to that

```
Client.limit(
5
).offset(
30
)
```

will return instead a maximum of 5 clients beginning with the 31st. The SQL looks like:

```
SELECT

*
FROM

clients LIMIT 5 OFFSET 30
```

6 Group

To apply a `GROUP BY` clause to the SQL fired by the finder, you can specify the `group` method on the find.

For example, if you want to find a collection of the dates orders were created on:

```
Order.select(
"date(created_at) as ordered_date, sum(price) as total_price"
).group(
"date(created_at)"
)
```

And this will give you a single `Order` object for each date where there are orders in the database.

The SQL that would be executed would be something like this:

```
SELECT

date
(created_at)
as

ordered_date,
sum
(price)
as

total_price
FROM

orders
GROUP

BY

date
(created_at)
```

7 Having

SQL uses the `HAVING` clause to specify conditions on the `GROUP BY` fields. You can add the `HAVING` clause to the SQL fired by the `Model.find` by adding the `:having` option to the find.

For example:

```
Order.select(
"date(created_at) as ordered_date, sum(price) as total_price"
).group(
"date(created_at)"
).having(
"sum(price) > ?"
,
100
)
```

The SQL that would be executed would be something like this:

```
SELECT

date
(created_at)
as

ordered_date,
sum
(price)
as

total_price
FROM

orders
GROUP

BY

date
(created_at)
HAVING
```

```
sum
(price) > 100
```

This will return single order objects for each day, but only those that are ordered more than $100 in a day.

8 Overriding Conditions

8.1 `except`

You can specify certain conditions to be excepted by using the `except` method. For example:

```
Post.where(
'id > 10'
).limit(
20
).order(
'id asc'
).except(
:order
)
```

The SQL that would be executed:

```
SELECT

*
FROM

posts
WHERE

id > 10 LIMIT 20
```

8.2 `only`

You can also override conditions using the `only` method. For example:

```
Post.where(
'id > 10'
).limit(
20
).order(
'id desc'
).only(
:order
,
:where
)
```

The SQL that would be executed:

```
SELECT

*
FROM

posts
WHERE

id > 10
ORDER

BY

id
DESC
```

8.3 `reorder`

The `reorder` method overrides the default scope order. For example:

```
class

Post < ActiveRecord::Base

..

..

has_many
:comments
,
:order

=>
'posted_at DESC'
end

Post.find(
10
).comments.reorder(
'name'
)
```

The SQL that would be executed:

```
SELECT

*
FROM

posts
WHERE

id = 10
ORDER

BY

name
```

In case the `reorder` clause is not used, the SQL executed would be:

```
SELECT

*
FROM

posts
WHERE

id = 10
ORDER

BY

posted_at
DESC
```

8.4 `reverse_order`

The `reverse_order` method reverses the ordering clause if specified.

```
Client.where(
"orders_count > 10"
).order(
:name
).reverse_order
```

The SQL that would be executed:

```
SELECT

*
FROM

clients
WHERE

orders_count > 10
ORDER

BY

name

DESC
```

If no ordering clause is specified in the query, the `reverse_order` orders by the primary key in reverse order.

```
Client.where(
"orders_count > 10"
).reverse_order
```

The SQL that would be executed:

```
SELECT

*
FROM

clients
WHERE

orders_count > 10
ORDER

BY

clients.id
DESC
```

This method accepts **no** arguments.

9 Readonly Objects

Active Record provides `readonly` method on a relation to explicitly disallow modification or deletion of any of the returned object. Any attempt to alter or destroy a readonly record will not succeed, raising an `ActiveRecord::ReadOnlyRecord` exception.

```
client = Client.readonly.first
client.visits +=
1
client.save
```

As `client` is explicitly set to be a readonly object, the above code will raise an `ActiveRecord::ReadOnlyRecord` exception when calling `client.save` with an updated value of *visits*.

10 Locking Records for Update

Locking is helpful for preventing race conditions when updating records in the database and ensuring atomic updates.

Active Record provides two locking mechanisms:

- Optimistic Locking
- Pessimistic Locking

10.1 Optimistic Locking

Optimistic locking allows multiple users to access the same record for edits, and assumes a minimum of conflicts with the data. It does this by checking whether another process has made changes to a record since it was opened. An `ActiveRecord::StaleObjectError` exception is thrown if that has occurred and the update is ignored.

Optimistic locking column

In order to use optimistic locking, the table needs to have a column called `lock_version`. Each time the record is updated, Active Record increments the `lock_version` column. If an update request is made with a lower value in the `lock_version` field than is currently in the `lock_version` column in the database, the update request will fail with an `ActiveRecord::StaleObjectError`. Example:

```
c1 = Client.find(
1
)
c2 = Client.find(
1
)

c1.first_name =
"Michael"
c1.save

c2.name =
"should fail"
c2.save
# Raises an ActiveRecord::StaleObjectError
```

You're then responsible for dealing with the conflict by rescuing the exception and either rolling back, merging, or otherwise apply the business logic needed to resolve the conflict.

You must ensure that your database schema defaults the `lock_version` column to `0`.

This behavior can be turned off by setting `ActiveRecord::Base.lock_optimistically = false`.

To override the name of the `lock_version` column, `ActiveRecord::Base` provides a class method called `set_locking_column`:

```
class

Client < ActiveRecord::Base

set_locking_column
:lock_client_column
end
```

10.2 Pessimistic Locking

Pessimistic locking uses a locking mechanism provided by the underlying database. Using `lock` when building a relation obtains an exclusive lock on the selected rows. Relations using `lock` are usually wrapped inside a transaction for preventing deadlock conditions.

For example:

```
Item.transaction
do

i = Item.lock.first

i.name =
'Jones'

i.save
end
```

The above session produces the following SQL for a MySQL backend:

```
SQL (0.2ms)
BEGIN
Item
Load

(0.3ms)
SELECT
```

```
*
FROM

`items` LIMIT 1
FOR

UPDATE
Item
Update

(0.4ms)
UPDATE

`items`
SET

`updated_at` =
'2009-02-07 18:05:56'
, `
name
` =
'Jones'

WHERE

`id` = 1
SQL (0.8ms)
COMMIT
```

You can also pass raw SQL to the `lock` method for allowing different types of locks. For example, MySQL has an expression called `LOCK IN SHARE MODE` where you can lock a record but still allow other queries to read it. To specify this expression just pass it in as the lock option:

```
Item.transaction
do

i = Item.lock(
"LOCK IN SHARE MODE"
).find(
1
)

i.increment!(
:views
)
end
```

If you already have an instance of your model, you can start a transaction and acquire the lock in one go using the following code:

```
item = Item.first
item.with_lock
do

# This block is called within a transaction,

# item is already locked.

item.increment!(
:views
)
end
```

11 Joining Tables

Active Record provides a finder method called `joins` for specifying `JOIN` clauses on the resulting SQL. There are multiple ways to use the `joins` method.

11.1 Using a String SQL Fragment

You can just supply the raw SQL specifying the `JOIN` clause to `joins`:

```
Client.joins(
'LEFT OUTER JOIN addresses ON addresses.client_id = clients.id'
)
```

This will result in the following SQL:

```
SELECT

clients.*
FROM

clients
LEFT

OUTER

JOIN

addresses
ON

addresses.client_id = clients.id
```

11.2 Using Array/Hash of Named Associations

This method only works with `INNER JOIN`.

Active Record lets you use the names of the **associations** defined on the model as a shortcut for specifying `JOIN` clause for those associations when using the `joins` method.

For example, consider the following `Category`, `Post`, `Comments` and `Guest` models:

```
class

Category < ActiveRecord::Base

has_many
:posts
end

class

Post < ActiveRecord::Base

belongs_to
:category

has_many
:comments

has_many
:tags
end

class

Comment < ActiveRecord::Base

belongs_to
:post

has_one
:guest
end

class

Guest < ActiveRecord::Base

belongs_to
:comment
end
```

```
class

Tag < ActiveRecord::Base

belongs_to
:post
end
```

Now all of the following will produce the expected join queries using `INNER JOIN`:

11.2.1 Joining a Single Association

```
Category.joins(
:posts
)
```

This produces:

```
SELECT

categories.*
FROM

categories

INNER

JOIN

posts
ON

posts.category_id = categories.id
```

Or, in English: “return a Category object for all categories with posts”. Note that you will see duplicate categories if more than one post has the same category. If you want unique categories, you can use Category.joins(:post).select(“distinct(categories.id)”).

11.2.2 Joining Multiple Associations

```
Post.joins(
:category
,
:comments
)
```

This produces:

```
SELECT

posts.*
FROM

posts

INNER

JOIN

categories
ON

posts.category_id = categories.id

INNER

JOIN

comments
ON

comments.post_id = posts.id
```

Or, in English: “return all posts that have a category and at least one comment”. Note again that posts with multiple comments will show up multiple times.

11.2.3 Joining Nested Associations (Single Level)

```
Post.joins(
:comments

=>
:guest
)
```

This produces:

```
SELECT

posts.*
FROM

posts

INNER

JOIN

comments
ON

comments.post_id = posts.id

INNER

JOIN

guests
ON

guests.comment_id = comments.id
```

Or, in English: “return all posts that have a comment made by a guest.”

11.2.4 Joining Nested Associations (Multiple Level)

```
Category.joins(
:posts

=> [{
:comments

=>
:guest
},
:tags
])
```

This produces:

```
SELECT

categories.*
FROM

categories

INNER

JOIN

posts
ON

posts.category_id = categories.id

INNER
```

```
JOIN

comments
ON

comments.post_id = posts.id

INNER

JOIN

guests
ON

guests.comment_id = comments.id

INNER

JOIN

tags
ON

tags.post_id = posts.id
```

11.3 Specifying Conditions on the Joined Tables

You can specify conditions on the joined tables using the regular **Array** and **String** conditions. **Hash conditions** provides a special syntax for specifying conditions for the joined tables:

```
time_range = (
Time
.now.midnight -
1
.day)..
Time
.now.midnight
Client.joins(
:orders
).where(
'orders.created_at'

=> time_range)
```

An alternative and cleaner syntax is to nest the hash conditions:

```
time_range = (
Time
.now.midnight -
1
.day)..
Time
.now.midnight
Client.joins(
:orders
).where(
:orders

=> {
:created_at

=> time_range})
```

This will find all clients who have orders that were created yesterday, again using a `BETWEEN` SQL expression.

12 Eager Loading Associations

Eager loading is the mechanism for loading the associated records of the objects returned by `Model.find` using as few queries as possible.

N + 1 queries problem

Consider the following code, which finds 10 clients and prints their postcodes:

```
clients = Client.limit(
10
)

clients.
each

do

|client|

puts client.address.postcode
end
```

This code looks fine at the first sight. But the problem lies within the total number of queries executed. The above code executes 1 (to find 10 clients) + 10 (one per each client to load the address) = **11** queries in total.

Solution to N + 1 queries problem

Active Record lets you specify in advance all the associations that are going to be loaded. This is possible by specifying the `includes` method of the `Model.find` call. With `includes`, Active Record ensures that all of the specified associations are loaded using the minimum possible number of queries.

Revisiting the above case, we could rewrite `Client.all` to use eager load addresses:

```
clients = Client.includes(
:address
).limit(
10
)

clients.
each

do

|client|

puts client.address.postcode
end
```

The above code will execute just **2** queries, as opposed to **11** queries in the previous case:

```
SELECT

*
FROM

clients LIMIT 10
SELECT

addresses.*
FROM

addresses

WHERE

(addresses.client_id
IN

(1,2,3,4,5,6,7,8,9,10))
```

12.1 Eager Loading Multiple Associations

Active Record lets you eager load any number of associations with a single `Model.find` call by using an array, hash, or a nested hash of array/hash with the `includes` method.

12.1.1 Array of Multiple Associations

```
Post.includes(
:category
,
:comments
)
```

This loads all the posts and the associated category and comments for each post.

12.1.2 Nested Associations Hash

```
Category.includes(
:posts

=> [{
:comments

=>
:guest
},
:tags
]).find(
1
)
```

This will find the category with id 1 and eager load all of the associated posts, the associated posts' tags and comments, and every comment's guest association.

12.2 Specifying Conditions on Eager Loaded Associations

Even though Active Record lets you specify conditions on the eager loaded associations just like `joins`, the recommended way is to use **joins** instead.

However if you must do this, you may use `where` as you would normally.

```
Post.includes(
:comments
).where(
"comments.visible"
,
true
)
```

This would generate a query which contains a `LEFT OUTER JOIN` whereas the `joins` method would generate one using the `INNER JOIN` function instead.

```
SELECT

"posts"
.
"id"

AS

t0_r0, ...
"comments"
.
"updated_at"

AS

t1_r5
FROM

"posts"

LEFT

OUTER

JOIN

"comments"

ON

"comments"
```

```
.
"post_id"

=
"posts"
.
"id"

WHERE

(comments.visible =
1
)
```

If there was no `where` condition, this would generate the normal set of two queries.

If, in the case of this `includes` query, there were no comments for any posts, all the posts would still be loaded. By using `joins` (an INNER JOIN), the join conditions **must** match, otherwise no records will be returned.

13 Scopes

Scoping allows you to specify commonly-used ARel queries which can be referenced as method calls on the association objects or models. With these scopes, you can use every method previously covered such as `where`, `joins` and `includes`. All scope methods will return an `ActiveRecord::Relation` object which will allow for further methods (such as other scopes) to be called on it.

To define a simple scope, we use the `scope` method inside the class, passing the ARel query that we'd like run when this scope is called:

```
class

Post < ActiveRecord::Base

scope
:published
, where(
:published

=>
true
)
end
```

Just like before, these methods are also chainable:

```
class

Post < ActiveRecord::Base

scope
:published
, where(
:published

=>
true
).joins(
:category
)
end
```

Scopes are also chainable within scopes:

```
class

Post < ActiveRecord::Base

scope
:published
, where(
:published

=>
true
)
```

```
scope
:published_and_commented
, published.
and
(
self
.arel_table[
:comments_count
].gt(
0
))
end
```

To call this `published` scope we can call it on either the class:

```
Post.published
# => [published posts]
```

Or on an association consisting of `Post` objects:

```
category = Category.first
category.posts.published
# => [published posts belonging to this category]
```

13.1 Working with times

If you're working with dates or times within scopes, due to how they are evaluated, you will need to use a lambda so that the scope is evaluated every time.

```
class

Post < ActiveRecord::Base

scope
:last_week
, lambda { where(
"created_at < ?"
,
Time
.zone.now ) }
end
```

Without the `lambda`, this `Time.zone.now` will only be called once.

13.2 Passing in arguments

When a `lambda` is used for a `scope`, it can take arguments:

```
class

Post < ActiveRecord::Base

scope :1_week_before, lambda { |time| where(
"created_at < ?"
, time) }
end
```

This may then be called using this:

```
Post.1_week_before(
Time
.zone.now)
```

However, this is just duplicating the functionality that would be provided to you by a class method.

```
class

Post < ActiveRecord::Base

def

self
.1_week_before(time)

where(
"created_at < ?"
, time)

end
end
```

Using a class method is the preferred way to accept arguments for scopes. These methods will still be accessible on the association objects:

```
category.posts.1_week_before(time)
```

13.3 Working with scopes

Where a relational object is required, the `scoped` method may come in handy. This will return an `ActiveRecord::Relation` object which can have further scoping applied to it afterwards. A place where this may come in handy is on associations

```
client = Client.find_by_first_name(
"Ryan"
)
orders = client.orders.scoped
```

With this new `orders` object, we are able to ascertain that this object can have more scopes applied to it. For instance, if we wanted to return orders only in the last 30 days at a later point.

```
orders.where(
"created_at > ?"
,
30
.days.ago)
```

13.4 Applying a default scope

If we wish for a scope to be applied across all queries to the model we can use the `default_scope` method within the model itself.

```
class

Client < ActiveRecord::Base

default_scope where(
"removed_at IS NULL"
)
end
```

When queries are executed on this model, the SQL query will now look something like this:

```
SELECT

*
FROM

clients
WHERE

removed_at
IS

NULL
```

13.5 Removing all scoping

If we wish to remove scoping for any reason we can use the `unscoped` method. This is especially useful if a `default_scope` is specified in the model and should not be applied for this particular query.

```
Client.unscoped.all
```

This method removes all scoping and will do a normal query on the table.

14 Dynamic Finders

For every field (also known as an attribute) you define in your table, Active Record provides a finder method. If you have a field called `first_name` on your `Client` model for example, you get `find_by_first_name` and `find_all_by_first_name` for free from Active Record. If you have a `locked` field on the `Client` model, you also get `find_by_locked` and `find_all_by_locked` methods.

You can also use `find_last_by_*` methods which will find the last record matching your argument.

You can specify an exclamation point (`!`) on the end of the dynamic finders to get them to raise an `ActiveRecord::RecordNotFound` error if they do not return any records, like `Client.find_by_name!("Ryan")`

If you want to find both by name and locked, you can chain these finders together by simply typing "`and`" between the fields. For example, `Client.find_by_first_name_and_locked("Ryan", true)`.

Up to and including Rails 3.1, when the number of arguments passed to a dynamic finder method is lesser than the number of fields, say `Client.find_by_name_and_locked("Ryan")`, the behavior is to pass `nil` as the missing argument. This is **unintentional** and this behavior will be changed in Rails 3.2 to throw an `ArgumentError`.

15 Find or build a new object

It's common that you need to find a record or create it if it doesn't exist. You can do that with the `first_or_create` and `first_or_create!` methods.

15.1 `first_or_create`

The `first_or_create` method checks whether `first` returns `nil` or not. If it does return `nil`, then `create` is called. This is very powerful when coupled with the `where` method. Let's see an example.

Suppose you want to find a client named 'Andy', and if there's none, create one and additionally set his `locked` attribute to false. You can do so by running:

```
Client.where(
:first_name

=>
'Andy'
).first_or_create(
:locked

=>
false
)
# => #<Client id: 1, first_name: "Andy", orders_count: 0, locked: false, created_at: "2011-08-30 06:09:2
```

The SQL generated by this method looks like this:

```
SELECT

*
FROM

clients
WHERE

(clients.first_name =
'Andy'
) LIMIT 1
BEGIN
INSERT
```

```
INTO

clients (created_at, first_name, locked, orders_count, updated_at)
VALUES

(
'2011-08-30 05:22:57'
,
'Andy'
, 0,
NULL
,
'2011-08-30 05:22:57'
)
COMMIT
```

`first_or_create` returns either the record that already exists or the new record. In our case, we didn't already have a client named Andy so the record is created and returned.

The new record might not be saved to the database; that depends on whether validations passed or not (just like `create`).

It's also worth noting that `first_or_create` takes into account the arguments of the `where` method. In the example above we didn't explicitly pass a `:first_name => 'Andy'` argument to `first_or_create`. However, that was used when creating the new record because it was already passed before to the `where` method.

You can do the same with the `find_or_create_by` method:

```
Client.find_or_create_by_first_name(
:first_name

=>
"Andy"
,
:locked

=>
false
)
```

This method still works, but it's encouraged to use `first_or_create` because it's more explicit on which arguments are used to *find* the record and which are used to *create*, resulting in less confusion overall.

15.2 `first_or_create!`

You can also use `first_or_create!` to raise an exception if the new record is invalid. Validations are not covered on this guide, but let's assume for a moment that you temporarily add

```
validates
:orders_count
,
:presence

=>
true
```

to your `Client` model. If you try to create a new `Client` without passing an `orders_count`, the record will be invalid and an exception will be raised:

```
Client.where(
:first_name

=>
'Andy'
).first_or_create!(
:locked

=>
false
)
# => ActiveRecord::RecordInvalid: Validation failed: Orders count can't be blank
```

As with `first_or_create` there is a `find_or_create_by!` method but the `first_or_create!` method is preferred for clarity.

15.3 `first_or_initialize`

The `first_or_initialize` method will work just like `first_or_create` but it will not call `create` but new. This means that a new model instance will be created in memory but won't be saved to the database. Continuing with the `first_or_create` example, we now want the client named 'Nick':

```
nick = Client.where(
:first_name

=>
'Nick'
).first_or_initialize(
:locked

=>
false
)
# => <Client id: nil, first_name: "Nick", orders_count: 0, locked: false, created_at: "2011-08-30 06:09:

nick.persisted?
# => false

nick.new_record?
# => true
```

Because the object is not yet stored in the database, the SQL generated looks like this:

```
SELECT

*
FROM

clients
WHERE

(clients.first_name =
'Nick'
) LIMIT 1
```

When you want to save it to the database, just call `save`:

```
nick.save
# => true
```

16 Finding by SQL

If you'd like to use your own SQL to find records in a table you can use `find_by_sql`. The `find_by_sql` method will return an array of objects even if the underlying query returns just a single record. For example you could run this query:

```
Client.find_by_sql("
SELECT

*
FROM
clients

INNER

JOIN

orders
ON

clients.id = orders.client_id

ORDER

clients.created_at desc")
```

`find_by_sql` provides you with a simple way of making custom calls to the database and retrieving instantiated objects.

17 `select_all`

`find_by_sql` has a close relative called `connection#select_all`. `select_all` will retrieve objects from the database using custom SQL just like `find_by_sql` but will not instantiate them. Instead, you will get an array of hashes where each hash indicates a record.

```
Client.connection.select_all(
"SELECT * FROM clients WHERE id = '1'"
)
```

18 `pluck`

`pluck` can be used to query a single column from the underlying table of a model. It accepts a column name as argument and returns an array of values of the specified column with the corresponding data type.

```
Client.where(
:active

=>
true
).pluck(
:id
)
# SELECT id FROM clients WHERE active = 1

Client.uniq.pluck(
:role
)
# SELECT DISTINCT role FROM clients
```

`pluck` makes it possible to replace code like

```
Client.select(
:id
).map { |c| c.id }
```

with

```
Client.pluck(
:id
)
```

19 Existence of Objects

If you simply want to check for the existence of the object there's a method called `exists?`. This method will query the database using the same query as `find`, but instead of returning an object or collection of objects it will return either `true` or `false`.

```
Client.exists?(
1
)
```

The `exists?` method also takes multiple ids, but the catch is that it will return true if any one of those records exists.

```
Client.exists?(
1
,
2
,
3
)
# or
Client.exists?([
1
,
2
,
3
])
```

It's even possible to use `exists?` without any arguments on a model or a relation.

```
Client.where(
:first_name

=>
'Ryan'
).exists?
```

The above returns `true` if there is at least one client with the `first_name` 'Ryan' and `false` otherwise.

```
Client.exists?
```

The above returns `false` if the `clients` table is empty and `true` otherwise.

You can also use `any?` and `many?` to check for existence on a model or relation.

```
# via a model
Post.any?
Post.many?

# via a named scope
Post.recent.any?
Post.recent.many?

# via a relation
Post.where(
:published

=>
true
).any?
Post.where(
:published

=>
true
).many?

# via an association
Post.first.categories.any?
Post.first.categories.many?
```

20 Calculations

This section uses count as an example method in this preamble, but the options described apply to all sub-sections.

All calculation methods work directly on a model:

```
Client.count
# SELECT count(*) AS count_all FROM clients
```

Or on a relation:

```
Client.where(
:first_name

=>
'Ryan'
).count
# SELECT count(*) AS count_all FROM clients WHERE (first_name = 'Ryan')
```

You can also use various finder methods on a relation for performing complex calculations:

```
Client.includes(
"orders"
).where(
:first_name

=>
'Ryan'
,
:orders

=> {
:status

=>
'received'
}).count
```

Which will execute:

```
SELECT

count
(
DISTINCT

clients.id)
AS

count_all
FROM

clients

LEFT

OUTER

JOIN

orders
ON

orders.client_id = client.id
WHERE

(clients.first_name =
'Ryan'

AND

orders.status =
'received'
)
```

20.1 Count

If you want to see how many records are in your model's table you could call `Client.count` and that will return the number. If you want to be more specific and find all the clients with their age present in the database you can use `Client.count(:age)`.

For options, please see the parent section, **Calculations**.

20.2 Average

If you want to see the average of a certain number in one of your tables you can call the `average` method on the class that relates to the table. This method call will look something like this:

```
Client.average(
"orders_count"
)
```

This will return a number (possibly a floating point number such as 3.14159265) representing the average value in the field.

For options, please see the parent section, **Calculations**.

20.3 Minimum

If you want to find the minimum value of a field in your table you can call the `minimum` method on the class that relates to the table. This method call will look something like this:

```
Client.minimum(
"age"
)
```

For options, please see the parent section, **Calculations**.

20.4 Maximum

If you want to find the maximum value of a field in your table you can call the `maximum` method on the class that relates to the table. This method call will look something like this:

```
Client.maximum(
"age"
)
```

For options, please see the parent section, **Calculations**.

20.5 Sum

If you want to find the sum of a field for all records in your table you can call the `sum` method on the class that relates to the table. This method call will look something like this:

```
Client.sum(
"orders_count"
)
```

For options, please see the parent section, **Calculations**.

21 Running EXPLAIN

You can run EXPLAIN on the queries triggered by relations. For example,

```
User.where(
:id

=>
1
).joins(
:posts
).explain
```

may yield

```
EXPLAIN for: SELECT `users`.* FROM `users` INNER JOIN `posts` ON `posts`.`user_id` = `users`.`id` WHERE
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | Extra       |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
|  1 | SIMPLE      | users | const | PRIMARY       | PRIMARY | 4       | const |    1 |             |
|  1 | SIMPLE      | posts | ALL   | NULL          | NULL    | NULL    | NULL  |    1 | Using where |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------------+
2 rows in set (0.00 sec)
```

under MySQL.

Active Record performs a pretty printing that emulates the one of the database shells. So, the same query running with the PostgreSQL adapter would yield instead

```
EXPLAIN for: SELECT "users".* FROM "users" INNER JOIN "posts" ON "posts"."user_id" = "users"."id" WHERE

QUERY PLAN
------------------------------------------------------------------------------

Nested Loop Left Join  (cost=0.00..37.24 rows=8 width=0)

Join Filter: (posts.user_id = users.id)

->  Index Scan using users_pkey on users  (cost=0.00..8.27 rows=1 width=4)

Index Cond: (id = 1)

->  Seq Scan on posts  (cost=0.00..28.88 rows=8 width=4)

Filter: (posts.user_id = 1)
(6 rows)
```

Eager loading may trigger more than one query under the hood, and some queries may need the results of previous ones. Because of that, `explain` actually executes the query, and then asks for the query plans. For example,

```
User.where(
:id

=>
1
).includes(
:posts
).explain
```

yields

```
EXPLAIN for: SELECT `users`.* FROM `users`  WHERE `users`.`id` = 1
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | users | const | PRIMARY       | PRIMARY | 4       | const |    1 |       |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
1 row in set (0.00 sec)

EXPLAIN for: SELECT `posts`.* FROM `posts`  WHERE `posts`.`user_id` IN (1)
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | posts | ALL  | NULL          | NULL | NULL    | NULL |    1 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
1 row in set (0.00 sec)
```

under MySQL.

21.1 Automatic EXPLAIN

Active Record is able to run EXPLAIN automatically on slow queries and log its output. This feature is controlled by the configuration parameter

```
config.active_record.auto_explain_threshold_in_seconds
```

If set to a number, any query exceeding those many seconds will have its EXPLAIN automatically triggered and logged. In the case of relations, the threshold is compared to the total time needed to fetch records. So, a relation is seen as a unit of work, no matter whether the implementation of eager loading involves several queries under the hood.

A threshold of `nil` disables automatic EXPLAINs.

The default threshold in development mode is 0.5 seconds, and `nil` in test and production modes.

Automatic EXPLAIN gets disabled if Active Record has no logger, regardless of the value of the threshold.

21.1.1 Disabling Automatic EXPLAIN

Automatic EXPLAIN can be selectively silenced with `ActiveRecord::Base.silence_auto_explain`:

```
ActiveRecord::Base.silence_auto_explain
do

# no automatic EXPLAIN is triggered here
end
```

That may be useful for queries you know are slow but fine, like a heavyweight report of an admin interface.

As its name suggests, `silence_auto_explain` only silences automatic EXPLAINs. Explicit calls to `ActiveRecord::Relation#explain` run.

21.2 Interpreting EXPLAIN

Interpretation of the output of EXPLAIN is beyond the scope of this guide. The following pointers may be helpful:

SQLite3: **EXPLAIN QUERY PLAN**

MySQL: **EXPLAIN Output Format**

PostgreSQL: **Using EXPLAIN**

Feedback

You're encouraged to help improve the quality of this guide.

If you see any typos or factual errors you are confident to patch, please clone **docrails** and push the change yourself. That branch of Rails has public write access. Commits are still reviewed, but that happens after you've submitted your contribution. **docrails** is cross-merged with master periodically.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Check the **Ruby on Rails Guides Guidelines** for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please **open an issue**.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the **rubyonrails-docs mailing list**.